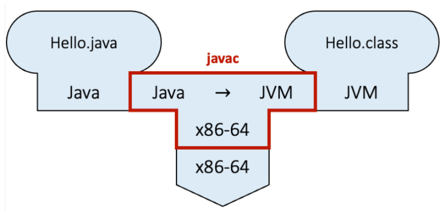
CS2030$_S$ Reference Notes

~github/reidenong/cheatsheets~, AY23/24 S1

1. Program and compiler

Java and JVM Tombstone diagram:



2. Variables and Types

Java Language

Java is a *statically* typed language, meaning variables must be declared with a type before they can be used.

Java is *strongly* typed, meaning that it enforces strict rules in its type system to ensure type safety.

Subtyping

A type `S` is a *subtype* of `T` if `S` can be used in any context where `T` is expected. This is represented by `S <: T`.
Whenever a supertype is needed, a subtype can be given.

Subtyping relationships are

Reflexive	for any `S`, `S <: S`
Transitive	`S <: T` and `T <: U` → `S <: U`
Anti-Symmetric	`S <: T` and `T <: S` → `S = T`

Primitive Subtyping

$$\begin{tikzcd}
\texttt{byte} \arrow[r] & \texttt{short} \arrow[r] & \texttt{int} \arrow[r] & \texttt{long} \arrow[r] & \texttt{float} \arrow[r] & \texttt{double} \\
& \texttt{char} \arrow[ur] & & & &
\end{tikzcd}$$

3. Functions

Abstraction Barriers enforce a separation of concerns between the implementer and client

5. Information Hiding

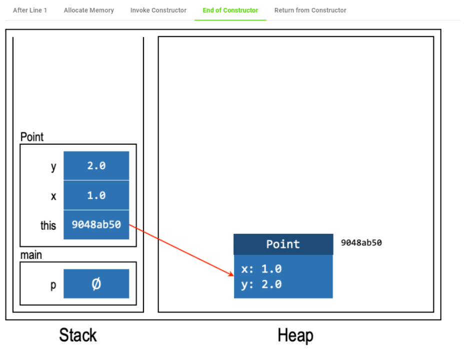
Information Hiding is violated when variables or information are exposed outside of the abstraction barrier

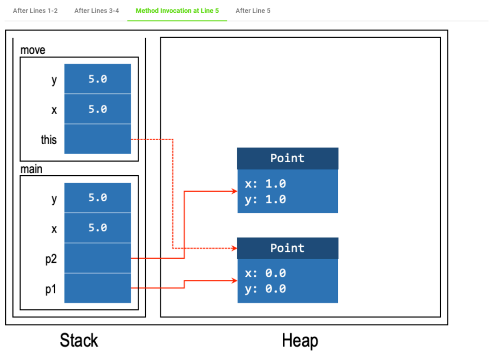
6. Tell Don't Ask

Occurs when entity A requests from another entity B and performs some action instead of telling B to do it itself

10. Stack and Heap diagrams

- Every `new` keyword means a new object is created in the heap
- Every method call means a new stackframe is created in the stack
- Every declaration means a new block in some method's stackframe
- Primitive type reassignment is shown as cancellation, ie. `int x = 1; x = 2;` is shown as `x` $= \cancel{1}\ 2$;





12. Method Overriding

Method Overriding is when a subclass provides a different implementation of a method from its superclass. It is a form of runtime polymorphism.

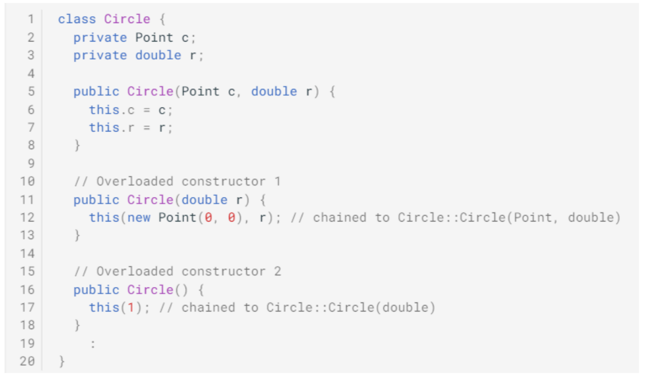
Method signatures include method name, number of parameters, type of parameters, order of parameters.

Method descriptors are the method signature including return type.

13. Method Overloading

Method Overloading is when a class has multiple methods with the same name but different signatures.

Overloading in Constructor Chaining:



```java
class Circle {
  private Point c;
  private double r;

  public Circle(Point c, double r) {
    this.c = c;
    this.r = r;
  }

  // Overloaded constructor 1
  public Circle(double r) {
    this(new Point(0, 0), r); // chained to Circle::Circle(Point, double)
  }

  // Overloaded constructor 2
  public Circle() {
    this(1); // chained to Circle::Circle(double)
  }
    :
}
```

14. Polymorphism

Polymorphism is the ability of an object to take on many forms, allowing us to perform a single action in different ways. This is done by defining one interface and having multiple implementations.

15. Method Invocation and Dynamic Binding

Dynamic Binding Process for `obj.foo(arg)`

Compile Time Step:

1. Determine `CTT(obj)` and `CTT(arg)`.
2. Determine all methods with the name foo in `CTT(obj)`.
   This includes all parent classes.
3. Determine all the methods from (2) that can accept `CTT(arg)` as an argument.
   > Correct number of parameters
   > Correct parameter types, ie. *supertype of* `CTT(arg)`
4. Determine the most specific method descriptor, else fail with compilation error.

Run Time Step:

1. Retrieve the method descriptor obtained from the *Compile Time Step*
2. Determine `RTT(obj)`.
3. Starting from `RTT(obj)`, search for the method descriptor obtained from (1).
   > If not found, search in parent classes.

Method Specificity

M is more specific than N if the arguments to M can be passed to N without compilation error.
ie. `equals(Integer)` is more specific than `equals(Object)`.

16. LSP

A subclass should not break the expectations set by the superclass.

Formally, if `S` is a subtype of `T`, then any object of type `T` can be replaced with an object of type `S` without changin the desirable property of the program.

final keyword

`final` prevents a class from being extended, or a method from being overridden.

17. Abstract Classes

An *abstract class* is a class that cannot be instantiated. It may have multiple fields and methods, where not all methods have to be abstract. An *abstract method* is a method that has no implementation (and thus no body).

A class with at least one abstract method must be declared abstract, but a abstract class can have no abstract methods.

18. Interfaces

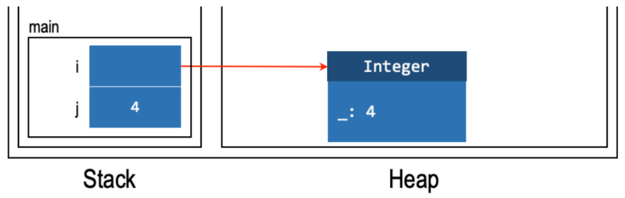
All methods in an interface are `public abstract`, and all fields are `public static final` by default. Interfaces may extend from one or more interfaces, but not classes.

19. Wrapper Classes

A class that encapsulates a type instead of field or methods.

`byte` → `Byte`	`short` → `Short`
`int` → `Integer`	`long` → `Long`
`float` → `Float`	`double` → `Double`
`char` → `Character`	`boolean` → `Boolean`

Stack and Heap of a wrapper class:



Autoboxing and Unboxing:

Auto Boxing and Unboxing is a single step process, and all wrapper classes have no subtyping relationships with each other.

```
Integer I = 4;    // Auto-boxing
int y = I;        // Auto-unboxing
```

Note that `Double d = 2` is an error, as Java cannot infer that `int` → `double` → `Double` is possible.

However, auto-unboxing can be followed by primitive subtyping, ie. the following code compiles.

```
Integer I = 4;
double d = I;
```

20. Run-Time Class Mismatch (Typecasting)

Typecasting can be used for narrowing type conversions from a superclass to a subclass, essentially asking the compiler to trust that the object is of the correct type.

Type Case Checks for casting `a = (C) b`

Compile Time Check:

1. Find `CTT(b)`
2. Check for the possibility* that `RTT(b)` is a subtype of `C`
   > if impossible, exit with compilation error
3. Find `CTT(a)`
4. Check if `C` is a subtype of `CTT(a)`
   > if not, exit with compilation error
   > otherwise, add run-time check for `RTT(b)` <: `C`

Run Time Check:

1. Find `RTT(b)`
2. Check if `RTT(b)` is a subtype of `C`
   > if not, exit with `ClassCastException`

Possibility Check
To check if it is possible for `RTT(b)` to be subtype of `C`

Case 1: `CTT(b)` is a subtype of `C`
This is widening and always allowed.

Case 2: `C` <: `CTT(b)`
This is narrowing and requires runtime checks.

Case 3: `C` is a interface.
There may be a subtype of `RTT(b)` that implements `C`, so this is allowed but subjected to runtime checks.

Impossible Case 1: `B` and `C` are unrelated.
It is impossible for `RTT(b)` to be a subtype of `C` as the subclass of `B` must already extend `B`. As such, it cannot extend from `C` as well as Java does not allow double inheritance.

Impossible Case 2: `C` is a interface and `B` is a `final` class. Then *Case 3* is impossible as there cannot exist a subtype of `CTT(b)` that implements `C`.

21. Variance

For a complex type `C`,

Covariant	`S <: T` → `C(S) <: C(T)`
Contravariant	`S <: T` → `C(T) <: C(S)`
Invariant	Neither Covariant nor Contravariant

Java arrays are covariant.

There are some run-time errors that cannot be detected by compiler when having covariant producers.
Consider `A1 <: B` and `A2 <: B`.

```
A1 aArr = new A1[] {new A1(), new A1()};

B[] bArr = aArr;
// Assume covariant: A1[] <: B[]

bArr[0] = new A2();
// Compiles because A2 <: B, but this is a
run-time error as bArr is actually an A1[]
and A2 </: A1
```

There are some run-time errors that cannot be detected by compiler when having contravariant consumers (hypothetical as Java is Covariant).
Consider `A1 <: B` and `A2 <: B`.

```
B[] bArr = new B[] {new A1(), new A2()};

A1[] aArr = bArr;
// Assume contravariant: B[] <: A1[]

A1 a1 = aArr[1];
// Compiles as A1 <: A1, but this is a run-
time error.
```

22. Exceptions

Unchecked Exceptions are exceptions caused by programmer error, and should not happen if perfect code is written. They are subclasses of `RuntimeException`, and examples include `NullPointerException`, `IllegalArgumentException`, `ClassCastException`.

Checked Exceptions are exceptions caused by external factors, and should be handled by the programmer. They are subclasses of `Exception`, and examples include `InputMismatchException`, `FileNotFoundException`.

When overriding a method that throws a checked exception, the overriding method must throw the same exception or a subclass of it.

Common `Exception` mistakes:
- DO NOT catch all exceptions with `Exception`.
- DO NOT exit the program within an exception, as it prevents the calling function from cleaning up its resources.
- DO NOT break the abstraction barrier, and handle implementation-specific exceptions within the barrier.
- DO NOT use exceptions as a control flow mechanism.

Java Errors Class
Java Errors are for situations where the program should terminate as there is no way to recover, ie. `OutOfMemoryError`, `StackOverflowError`. They share a common superclass with exceptions called `Throwable`, which has `getMessage():String` and `toString():String` methods.

Java Exception Examples
`ArrayStoreException` is thrown when an array is assigned with an incompatible type.

```
Object[] objArr = new Integer[10];
objArr[0] = "Hello"; // Throws
ArrayStoreException
```

`ClassCastException` is thrown when a typecast is performed on incompatible types.

```
Object obj = "Hello";
Integer i = (Integer) obj; // Throws
ClassCastException
```

23. Generics

Generics allow classes/methods that use any reference type to be defined without resorting to using `Object`. It enforces type safety by binding the generic type to a specific given type argument at compile time. *Note that primitive types cannot be passed as type parameters in Generics.*

Bounded Type parameters can be used to restrict the type of the generic type parameter, ie. `<T extends Number>` restricts the type parameter to be a subclass of `Number`.

Generic array declaration is fine but instantiation is not.

```
T[] arr;           // Declaration is ok
new Pair<T>[2];    // Instantiation is not
new T[2];          // Instantiation is not
```

24. Type Erasure

Essentially, all `< >` are removed from the code, and all type parameters are replaced with Object or a specified subtype if they are *bounded*. Note that Bounded type is limited to within brackets, ie. `<S extends X>` → `S` is erased to `X`.

Generics and arrays don't mix well together. Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that type.

```
// create a new array of pairs
Pair<String,Integer>[] pairArray;
pairArray = new Pair<String,Integer>[2];

// pass around the array of pairs as an array
of object
Object[] objArray = pairArray;

// put a pair into the array -- no
ArrayStoreException!
objArray[0] = new Pair<Double,Boolean>(3.14,
true);
```

```
// Heap pollution, now we get
ClassCastException
String str = pairArray[0].getFirst();
```

Arrays are *reifiable*, meaning full type information is available only at run-time. Generics are not reifiable due to type erasure, hence Java designers do not mix both.

25. Unchecked Warnings

Unchecked warnings occur when the compiler cannot guarantee type safety due to type erasure. If we are sure the code compiles with no issue and is type safe, we can dismiss it with `@SuppressWarnings` annotations.

Consider the folloing code in `Array<T>` class.

```
// Array<T> Constructor
public Array(int size) {
  /* The only way we can put an object into
array
   * is through set() and we only put object
of
   * type T inside. Hence it is safe to cast
   * Object[] to T[].
   */
  @SuppressWarnings("unchecked")
  T[] arr = (T[]) new Object[size];
  this.arr = arr;
}
```

`@SuppressWarnings` should only be used at the most limited scope to avoid unintentionally suppressing other valid concerns from the compiler.

Raw Types

Raw types are generic types without type arguments. Compiler cannot do any type checking, and code could bomb with `ClassCastException` at run-time.

Safe Varargs

`@SafeVarargs` is used to suppress unchecked warnings for varargs methods with generic types.

```
// Only items of type T goes into the array
@SafeVarargs
public static <T> ImmutableArray<T> of(T...
items) {
  // Do something
}
```

26. Wildcards

Wildcards are used to typecheck *generics with generics* as generics are invariant.

Upper Bounded Wildcards

Upper Bounded Wildcards are of the form `<? extends T>` to restrict the type parameter to be a subtype of `T`. They are Covariant, and is used for *Producers* when the method will need to receive input from all possible subtypes, ie. the `Array::copyFrom` method.

Lower Bounded Wildcards

Lower Bounded Wildcards are of the form `<? super T>` to restrict the type parameter to be a supertype of `T`. They are Contravariant, and is used for *Consumers* when the method will need to output to all possible supertypes, ie. the `Array::copyTo` method.

PECS

PECS is from the collection's point of view, if pulling items from a generic collection, it is a Producer and should `extend`; if pushing items into a generic collection, it is a Consumer and should `super`. If doing both, just use `<T>` with no wildcards.

Unbounded Wildcards

`<?>` is the supertype of every parameterized type of `<T>`. These are useful to write in methods to take in an `Array` of any type, or for declaration of fields, ie.

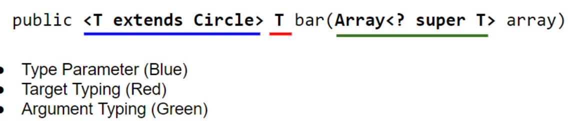
```
new Comparable<?>[10];
// Array of any type that implements
Comparable
Pair<?,?> p;
// Declaration is ok to take in pair of any
type.
```

The following code illustrates type safety with `<?>`

```
void foo(Array<?> arr) {
    x = arr.get(0);     // x must be Object
    arr.set(0, y);      // y must be null
}
```

27. Type Inference

Type inference and the diamond operator `< >` means Java can infer the RHS instantiation type from the LHS declaration type.



```
public <T extends Circle> T bar(Array<? super T> array)
```

- Type Parameter (Blue)
- Target Typing (Red)
- Argument Typing (Green)

Given `Type1` <: `T` <: `Type2`, Java infers `Type1`.

28. Immutability

A class is *immutable* when there cannot be any visible changes outside of its abstraction barrier.

Immutability Checklist:

- All fields are `final` (not necessarily)
- Type of all fields are immutable
- Arrays are copied before assignment
- No mutators
- Class has `final` modifier

29. Nested Classes

Nested Classes are used to group logically relevant classes together. They can be Inner Classes, Static Nested Classes, Local Classes, or Anonymous Classes.

Inner Classes

Non-static, and inside another class. It can access both static and non-static contexts.

Static Nested Classes

Static, and inside another class. It can only access static contexts, ie. other Class fields and Class methods.

Local Classes

Non-static, and inside a method. It can access both static and non-static contexts.

```
void sortNames(List<String> names) {
  class NameComparator implements
Comparator<String> {
    public int compare(String s1, String s2)
{
      return s1.length() - s2.length();
    }
  }
  names.sort(new NameComparator());
}
```

Variable Capture in Local Classes

When methods return, all local variables of the method are removed from the stack. However, a instance of a local class might exist.

As such, local classes make a copy of the local variables declared in the enclosing method. It only captures variables (i) local to the method and (ii) used in the local class.

Local classes are only allowed to access variables that are `final` or effectively `final`. However, nothing can be done about mutation and aliasing of reference types.

Anonymous Classes

with the format `new X (args) {body}`, where `X` is a class that the anonymous class extends or an interface that the anonymus class implements. `args` are to be passed into the constructor of the anonymous class.

```
names.sort(new Comparator<String>() {
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
});
```

30. Side Effect-Free Programming

A function is *pure* if it has no side effects. It is *referentially transparent* if it can be replaced with its return value without affecting the program.

Lambda Expressions

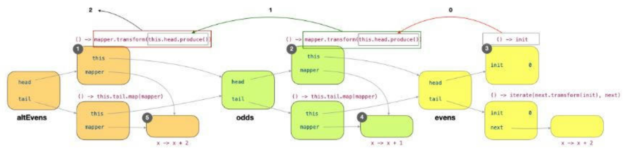
```
A::f     // x -> A.f(x)
A::new   // x -> new A(x)
a::g     // x -> a.g(x)

A::h
// (x, y) -> x.h(y) or (x, y) -> A.h(x, y)
```

Consider the last example, interpretation depends on how many parameters `h` takes and whether `h` is a class method or instance method.

33. InfiniteList

Sample Heap diagram for InfiniteList:

```
evens.map(x -> x + 1)
     .map(x -> x * 2)
     .head();
```


2
1
0
() -> mapper.transform(this.head.produce())
() -> mapper.transform(this.head.produce())
() -> init
1
this
mapper
2
this
mapper
3
init 0
head
tail
altEvens
() -> this.tail.map(mapper)
this
mapper
5
x -> x + 2
head
tail
odds
() -> this.tail.map(mapper)
this
mapper
4
x -> x + 1
head
tail
evens
() -> iterate(next.transform(init), next)
init 0
next
x -> x + 2

34. Streams in Java

Useful Functional Interfaces

`BooleanCondition<T>::test` ↔ `Predicate<T>::test`
`Producer<T>::produce` ↔ `Supplier<T>::get`
`Consumer<T>::consume` ↔ `Consumer<T>::accept`
`Transformer<T, R>::transform` ↔ `Function<T, R>::apply` / `UnaryOp<T>::apply`

Stream Creation

1. `Stream<T> Stream::of(T...)`
2. `Stream<T> Stream::generate(Supplier<T>)`
3. `Stream<T> Stream::iterate(T, UnaryOp<T>)`

Intermediate Operations

1. `filter(Predicate<T>)`
2. `map(Function<T, R>)`
3. `flatMap(Function<T, Stream<R>>)`
5. `distinct()`
6. `sorted()`
7. `peek(Consumer<T>)`
8. `limit(long)`
9. `skip(long)`

Terminal Operations

1. `anyMatch(Predicate<T>)` : `boolean`
2. `allMatch(Predicate<T>)` : `boolean`
3. `noneMatch(Predicate<T>)` : `boolean`
4. `collect()` Collects the elements of the stream into a collection, eg. `s.collect(Collectors.toList())`
5. `count()` : `long`
6. `findAny()` Returns a single element from the stream
7. `findFirst()` : `Optional<T>` Returns the first element from the stream
8. `forEach(Consumer<T>)` : `void`
9. `min((x,y) -> x.compareTo(y))` : `Optional<T>`
10. `max((x,y) -> x.compareTo(y))` : `Optional<T>`
11. `reduce(init, (subtotal, elem) -> ?)` : `T`
12. `toArray()` : `Object[]`

36. Monad

Three Monad Laws

1. Left Identity Law:

`Monad.of(x).flatMap(x -> f(x))` ≡ `f(x)`

2. Right Identity Law:

`m.flatMap(x -> Monad.of(x))` ≡ `m`

2. Associative Law

`m.flatMap(x -> f(x)).flatMap(y -> g(y))` ≡
`m.flatMap(x -> f(x).flatMap(y -> g(y)))`

Two Functor Laws

A functor is a abstraction in FP which ensures lambdas can be applied sequentially to the value without worrying about side information.

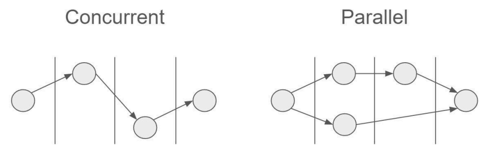
1. Identity Law:

`funct.map(x -> x)` ≡ `funct`

2. Composition Law:

`funct.map(x -> f(g(x)))` ≡ `funct.map(g).map(f)`

37. Parallel Streams



All parallels programs are concurrent.

Parallelism with Java streams `.parallelStream()`

`.parallel()` in streams breaks down the stream into subsequences to process. There is no coordination among parallel tasks, and the order of processing is not guaranteed.

When a parallel task is stateless, with no side effects, and each element is processed individually, the computation is known as *embarrassingly parallel.*

For a task to be parallelized, stream operations must not (1) Interfere with stream data, and should be (2) Stateless.

Interference means that one of the stream operations modifies the source of the stream during the execution of the terminal operation.

```
list.stream()
    ...
    .peek(() -> list.add(1)) // Interference
    // throws ConcurrentModificationException
```

A stateful lambda is one where the result depends on any state that might change during the execution of the stream.

Side effects can arise from usage of *non-thread-safe* data structures such as `ArrayList`. This can be resolved by using `.collect(Collectors.toList())` or `CopyOnWriteArrayList` or `toList()`.

Associativity with `.reduce()`

Versions of `.reduce()`:

```
.reduce(BinaryOperator<T> accumulator);
// returns Optional<T>

.reduce(T identity,
        BinaryOperator<T> accumulator);
// returns T

.reduce(U identity,
        BiFunction<U, ? super T, U>
          accumulator,
        BinaryOperator<U> combiner);
        // parallel
// returns U
```

Rules:

1. `combiner.apply(identity, u)` ≡ `u`
2. `combiner` and `accumulator` must be associative such that order of application should not matter, ie. `(t1 op t2) op t3` ≡ `t1 op (t2 op t3)`
3. `combiner` and `accumulator` must be compatible, ie. `combiner` must be able to accept the output of `accumulator`, ie.

`combiner.apply(u, accumulator.apply(identity, t))` ≡ `combiner.apply(u, t)`

Ordered vs Unordered Source

Streams created from `iterate`, ordered collections from `.of()` are ordered. Streams created from `generate` or unordered collections are unordered.

Stream operations such as `.distinct()` and `.sorted()` are stable by ordering. Meanwhile the parallel version of `.findFirst()`, `.skip()` and `.limit()` are expensive on ordered streams due to the need for inter stream coordination, and can be sped up by `.unordered()` if possible.

39. Asynchronous Programming

Monadic functions combines program fragments that return data with context.

Creating a `CompletableFuture`

1. using the `completedFuture(T)` method, which is a task that is already completed and return a value.
2. using the `runAsync(Runnable)` method, which is a task that runs asynchronously and returns nothing.
3. using the `supplyAsync(Supplier<T>)` method, which is a task that takes in a `Supplier<T>` lambda, runs asynchronously and returns a `CompletableFuture<T>`.

Chaining `CompletableFuture`

1. `thenApply` ↔ `map`
2. `thenCompose` ↔ `flatMap`
3. `thenCombine` ↔ `combine`
4. `thenRun` takes in a `Runnable`, and executes it after the current stage is completed.
5. `runAfterBoth` takes in another `CompletableFuture` and a `Runnable`, and executes it after both CompletableFutures are completed.
6. `runAfterEither` takes in another `CompletableFuture` and a `Runnable`, and executes it after either CompletableFuture is completed.

These methods run inthe same thread as the caller. Asynchronous versions are `thenApplyAsync`, `thenComposeAsync`, `thenCombineAsync` etc., and cause the lambda to run in a different thread.

Getting the Result

1. `CompletableFuture<T>::get()` is a synchronous call blocks the current thread until the result is available. It throws checked exceptions `InterruptedException` and `ExecutionException` which need to be caught and handled.
2. `CompletableFuture<T>::join()` is similar but without checked exceptions.

The `.handle(BiFunction)` method can be used to handle exceptions, with the first parameter being the value, the second the exception, and the third as the return value.

```
cf.thenApply(x -> x + 1)
  .handle((t,e) -> (e == null) ? t : 0)
  .join();
```

40. Fork and Join

We can create a task that we can fork and join as an instance of abstract class `RecursiveTask<T>`. This class has a abstract method `compute()` that returns a value of type `T`.

```
class Summer extends RecursiveTask<Integer> {
  ...

  @Override
  protected Integer compute() {
    // stop splitting into subtask if array
is already small.
    if (high - low < FORK_THRESHOLD) {
      ...
    }

    int middle = (low + high) / 2;
    Summer left = new Summer(low, middle,
array);
    Summer right = new Summer(middle, high,
array);
    left.fork();
    return right.compute() + left.join();
    // Order is important, do not cross!!!
  }
}

// in main()
Summer task = new Summer(0, array.length,
array);
int sum = task.compute();
```

ForkJoinPool

- `fork()` makes the caller add itself to the head of the deque of the thread and returns immediately.
- `join()` calls `.compute()` if the subtasks have not been completed, and returns the result.
- when a thread is idle, it checks its deque of tasks, and either picks up a task at the head to compute, or steals a task from the tail of another thread's deque.